(2)

RL-TR-91-46
In-House Report
March 1991

A USER'S GUIDE TO THE TEXPLAN SYSTEM

Michele Kubis and Colleen A. McAuliffe

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RL-TR-91-46 has been reviewed and is approved for publication.

APPROVED:

[illegible]

SAMUEL A. DINITTO, JR., Chief
C^2 Software Technology Division
Directorate of Command and Control

APPROVED:

[illegible]

Raymond P. Urtz, Jr.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:

[illegible]

RONALD RAPOSO
Directorate of Plans and Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
March 1991

3. REPORT TYPE AND DATES COVERED
In-House Jun 90 - Sep 90

4. TITLE AND SUBTITLE
A USER'S GUIDE TO THE TEXPLAN SYSTEM

5. FUNDING NUMBERS
PE - 62702F
PR - 5581
TA - 27
WU - 30

6. AUTHOR(S)
Michele Kubis and Colleen A. McAuliffe

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rome Laboratory (COES)
Griffiss AFB NY 13441-5700

8. PERFORMING ORGANIZATION REPORT NUMBER
RL-TR-91-46

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Douglas A. White/COES/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report is a guide to the use of the TEXPLAN (Textual EXplanation PLANer) natural language generation system in its application to a knowledge-based battle simulation system. TEXPLAN is a natural language text generator which composes single and multiparagraph texts and can be logically divided into two major components: a linguistic realization component which realizes an explanation as English text, and a test planner which plans the content of the explanation. The purpose of this report is to provide details of these two components, dissecting them into different sections, then describing the underlying functions which constitute each section.

14. SUBJECT TERMS
Natural Language Generation, Text Planning, Artificial Intelligence, User Interface

15. NUMBER OF PAGES
56

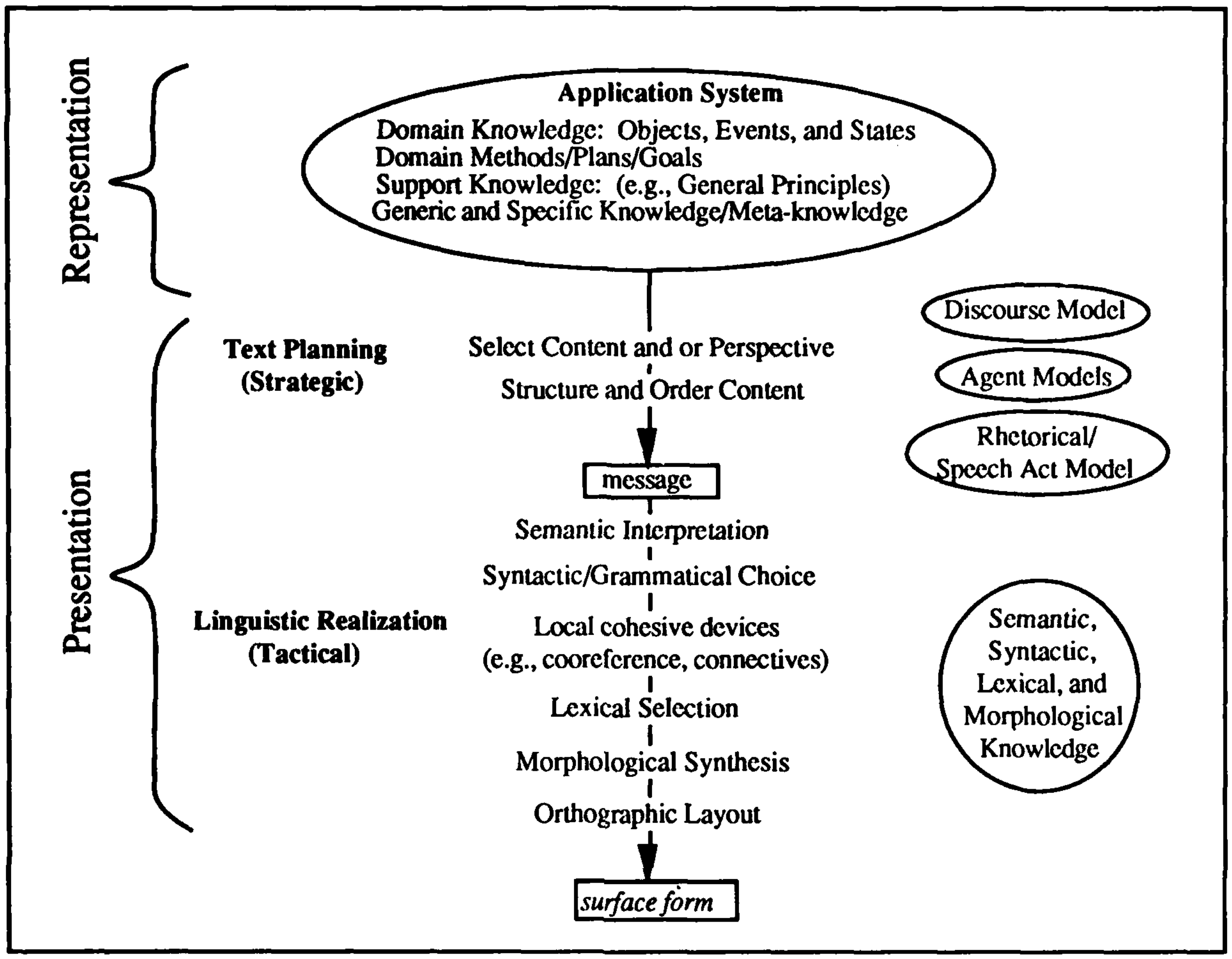
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

TABLE OF CONTENTS



Representation
Application System
Domain Knowledge: Objects, Events, and States
Domain Methods/Plans/Goals
Support Knowledge: (e.g., General Principles)
Generic and Specific Knowledge/Meta-knowledge
Presentation
Text Planning (Strategic)
Select Content and or Perspective
Structure and Order Content
message
Semantic Interpretation
Syntactic/Grammatical Choice
Linguistic Realization (Tactical)
Local cohesive devices (e.g., coorefcrence, connectives)
Lexical Selection
Morphological Synthesis
Orthographic Layout
surface form
Discourse Model
Agent Models
Rhetorical/ Speech Act Model
Semantic, Syntactic, Lexical, and Morphological Knowledge

TEXPLAN DIAGRAM

This diagram illustrates the different components of the TEXPLAN natural language generation system. The representation of the application system consists of domain knowledge, domain methods, plans and goals, and generic and specific knowledge. The presentation bracket divides TEXPLAN into two components: text planning and linguistic realization. The text planning component plans what to say and how to say it, based on the specific goals to be realized and the models to be used. The linguistic realization component involves the organization and interpretation of information gathered into specific messages. These messages are constructed using rhetorical predicates which are chosen by the planning component of the system to realize its goals. The linguistic component organizes the messages through semantic interpretation and syntactic choices. It also chooses cohesive devices (words to clarify goals), and synthesizes the proper surface structures of words, eventually leading up to the final surface form, generated English text.

INTRODUCTION

TEXPLAN (Textual Explanation Planner) is a natural language text generator which composes single and multi-paragraph texts (Maybury, 1990c). TEXPLAN can be logically divided into two major components: a linguistic realization component which realizes an explanation as English text, and a text planner which plans the content of the explanation. The purpose of this report is to provide details of these two components, dissecting them into different sections and then describing the underlying functions which constitute each section.

TEXPLAN has been ported to LACE (Land Air Combat in ERIC), a knowledge-based battle simulation system (Anken, 1989) written in the programming language ERIC (Hilton, 1987). TEXPLAN uses special accessor functions written in ERIC to retrieve information about the LACE application. This information is represented in LACE as an object-oriented frame-like language. The LACE system generates this information about its missions and objects from the Route Planner, the Cartographic database (Hilton and Anken, 1990) and the ORACLE database (Knowledge Systems Concept, Inc., 1989). The Route Planner contains information relating to the missions (e.g., offensive-counter-air-missions), the Cartographic database contains information about the generated map-objects (e.g., towns, airstrips and lakes) and the ORACLE database contains information about targets such as air-facilities, aircraft and sam-sites (surface-to-air-missile sites).

TEXPLAN uses a message, represented in a list, to determine the organization of the components of a sentence. This list is organized to include subject(s), articles, direct objects, indirect objects and modifiers. Each component has a specific position in the message, allowing the sentence generator to recognize the usage of a word based on its position in the structure.

To generate grammatical text, TEXPLAN uses an on-line dictionary and a representation of English syntax and semantics. It is also able to create entries for words that are not already in its dictionary. For example, there are thousands of objects in LACE. Since listing them in the dictionary would be inefficient and time-consuming, the dictionary automatically generates entries for them. This ability to create entries increases TEXPLAN's versatility, since it can use words that do not exist in the dictionary file. It uses specific functions that retrieve or create word entries, or that construct different types of entries for different types of objects. The lexical entries in the dictionary are the components of the message lists which are manipulated to produce grammatical English.

At the paragraph level, TEXPLAN can describe and compare entities, and plan directions to and from entities in the LACE application. TEXPLAN combines different types of sentences, using a library of plan operators, to form groups of text that realize specific goals. Individual goals are actualized through a selection of rhetorical predicates. A rhetorical predicate (rp) conveys specific information about an object or objects in the LACE application. For example, there are rhetorical predicates that define entities, describe their attributes, and so on. Each rp corresponds to one sentence. Using the plan operators, TEXPLAN reasons about how best to organize these rhetorical predicates to generate an effective text.

This report is structured as follows. First it discusses the linguistic realization component, which includes the different kinds of rhetorical predicates, the sentence-level syntax and semantics, and the dictionary. The report then discusses the text planning component, and how TEXPLAN produces descriptive and comparative texts as well as route directions. The report concludes by indicating future directions which include the generation of reports about events in the LACE simulation.

SECTION 1

RHETORICAL PREDICATES

The first step in understanding how TEXPLAN works is to become familiar with all of the system's rhetorical predicates. This section will acquaint the reader with all of the rhetorical predicates used by TEXPLAN. The predicates are used to generate different types of sentences which contain different types of information. They form messages or lists which organize this information. These messages are then manipulated by the other functions in the system until the final generated text is realized.

There are some functions that are important to a majority of the predicates in the system. *Relevant-attributes* returns the most important and the most informative attributes (characteristics) of the given entity. It calls the function *relevant-attributes-1*, which uses a list of what is irrelevant along with a list of all of the entity's slots to find the relevant attributes. *Relevant-attributes-2* is also used by several functions and is identical to *relevant-attributes*, except *relevant-attributes-2* does not include location and roads as relevant attributes. *Attribute-value-pairs* is also used by many of the predicates. It returns the attribute properties and their values in a predicate list structure.

The format for this section is as follows: name the predicate, describe what it does and how it does it (including a description of any important functions that it calls), list the message and give an example of the type of sentence(s) that can be produced when the predicate is used.

Definition

The *definition* predicate is one of the most basic predicates in TEXPLAN. Its output is easy to generate; *definition* simply defines what class a specific entity is from.

Example

Message:

```
(DEFINITION
    ((#<TOWN GRIMMA>))
    ((#<CLASS TOWN>)))
```

Text: *Grimma is a town.*

Logical-definition

Logical-definition is more descriptive than the definition predicate. It not only describes what the entity is, but it also describes all of the distinctive attributes of the entity, using the function *differentia-attributes.*

Example

Message:

```
(LOGICAL-DEFINITION
    ((#<TOWN FREIBERG>))
    ((#<CLASS TOWN>))
    NIL
    NIL
    ((LOCATION NIL INDEF ((13 DEGREES-LONGITUDE)
                          (51 DEGREES-LATITUDE)))
     (STATUS NIL INDEF FULLY-FUNCTIONAL)
     (POPULATION NIL INDEF 25000-TO-100000)
     (ROADS 2 NIL
      (#<road-segment 173 33UUS8441 33UUS8041>
       #<road-segment 101 33UUS8440 33UUS8441>
       #<road-segment 101 33UUS8441 33UUS8046>
       #<road-segment 173 33UUS9049 33UUS8441>))))
```

Text: *Freiberg is a town with a location of thirteen degrees longitude and fifty-one degrees latitude, a functional-status of fully-functional, a population of 25000-to-100000, and two roads (Route 101 and Route 173).*

Compare-logical-definition

Compare-logical-definition is exactly like logical-definition, however compare-logical-definition is used when comparing two entities. The two entities are tested by the function *compare-logic-def* for similarity. Depending on their relationship, either the word "equivalently", "similarly", or "in-contrast" is used in the final surface form at the beginning of the sentence.

Example 1

Message:

```
(COMPARE-LOGICAL-DEFINITION
    ((#<TOWN ERFURT>))
    ((#<CLASS TOWN>))
    NIL
    NIL
    ((SIMILARLY)
     (LOCATION NIL INDEF ((11 DEGREES-LONGITUDE)
                          (51 DEGREES-LATITUDE)))
     (STATUS NIL INDEF FULLY-FUNCTIONAL)
     (POPULATION NIL INDEF 100000-OR-MORE)
     (ROADS 5 NIL (#<road-segment...))))
```

Text: *Similarly, Erfurt is a town with a location of eleven degrees longitude and fifty-one degrees latitude, a functional-status of fully-functional, a population of 100000-or-more, and five roads (Route 7f, Route 5f, Route 274f, Route 7, and Route 4).*

Example 2

Message:

```
(COMPARE-LOGICAL-DEFINITION
    ((#<CLASS MIG-27>))
    ((#<CLASS FIGHTER>))
    NIL
    NIL
    ((IN-CONTRAST)
     (NICK-NAME NIL INDEF ((FLOGGER-D)))
     (COMRAD NIL INDEF ((1380.0 KILOMETERS)))
     (MAXSPD NIL INDEF NIL) (MAXWT NIL INDEF NIL)
     (CRUISE-SPEED NIL INDEF ((490 METERS-PER-SECOND)))
     (A-FACTOR NIL INDEF VERY-WEAK)
     (B-FACTOR NIL INDEF STRONG)
     (MAXFUEL NIL INDEF ((2000 UNITS)))
     (METERS-PER-POUND-FUEL 233.0)
     (MAX-SPEED NIL INDEF ((5 METERS-PER-SECOND)))))
```

Text: *In contrast, a MIG-27 is a fighter with a nick-name of Flogger-D, combat radii of 1,380 kilometers, a maximum speed capacity, a maximum weight, a cruise-speed of 490 meters-per-second, an offensive capability of very-weak, a defensive capability of strong, a maximum fuel capacity of 2,000 units, 233.0 meters per pound of fuel, and a maximum speed of five meters-per-second.*

Synonymic-definition

Synonymic-definition tests to see if an entity has a code-name or a nick-name in the underlying application. If it does, the function *synonymic* will find the name and return it. If it does not, it will say that the entity has no code name.

Example 1

Message:
```
(SYNONYMIC-DEFINITION
   ((#<CLASS MIG-29>))
   ((NICK-NAME NIL INDEF ((FULCRUM)))))
```

Text: *A MIG-29 has a nick-name of Fulcrum.*

Example 2

Message:
```
(SYNONYMIC-DEFINITION
   ((#<TOWN FREIBERG>))
   ((CODE-NAME NO NO-ARTICLE)))
```

Text: *Freiberg has no code name.*

Attributive

The *attributive* predicate describes the attributes (i.e. the characteristics or properties) of an entity. Its primary function is *attributes*, which returns the relevant attributes of the entity.

Example

Message:
```
(ATTRIBUTIVE
   ((#<CLASS LAND-ROLL>))
   ((FREQUENCY 4 INDEF NIL NIL NIL (G H I J))
    (VERTICAL-BEAMWIDTH NIL INDEF ((1 DEGREE)))
    (HORIZONTAL-BEAMWIDTH NIL INDEF ((3 DEGREES)))
    (MAP-DISPLAY NIL INDEF NIL)))
```

Text: *A land-roll has four band designators (g, h, i, and j), a vertical beamwidth of one degree, a horizontal bandwidth of three degrees, and a map-display.*

Elaboration

This predicate describes the attributes of an entity that are not of major importance. It calls the function *elaborative-attribute-value-pairs* to return these attributes.

Example

Message:

```
(ELABORATION
    ((#<LAKE WIPPER-TALSPERRE> #<CLASS LAKE>))
    ((BLOCK NIL INDEF #<BLOCK 32UPC5010>)
     (NAME NIL INDEF ((“WIPPER-TALSPERRE”)))
     (PERIMETER 3 INDEF NIL NIL NIL
            (32UPC5216 32UPC5315 32UPC5214))))
```

Text: *The lake Wipper-Talsperre has a block of Block 32upc5010, a name of wipper-talsperre, and three perimeters (32upc5216, 33upc5315, and 32upc5214).*

Constituent

Constituent describes the constituents of an entity. It calls the function *constituents*, which specifies what attributes can be considered constituents (i.e. parts or subparts) of an entity.

Example 1

Message:

```
(CONSTITUENT
    ((#<TOWN EILENBURG> #<CLASS TOWN>))
    ((ROADS 2 NIL (#<road-segment...))))
```

Text: *The town Eilenburg has two roads (Route 107 and Route 87).*

Example 2

Message:

```
(CONSTITUENT
    ((#<AIRSTRIP BRANDIS> #<CLASS AIRSTRIP>))
    ((RUNWAY 2 INDEF NIL NIL NIL (33UUS3689 33UUS3889))))
```

Text: *The airstrip Brandis has two runways (33uus3689 and 33uus3889).*

Classification

This function returns the offspring (children or next lowest level in the hierarchy) of a class object. It does not work on instances, since they do not have offspring (instances constitute the lowest level in the hierarchy).

Example

Message:
```
((#<CLASS LAKE>)) ((INSTANCES-OF 39 ARTICLE))
    NIL
    ((#<LAKE HRACHOLUSKY>) (#<LAKE BECKEN>)...)))
```

Text: *There are thirty-nine instances of lakes: Hracholusky, Becken,...*

Location

The *location* predicate retrieves the latitude and longitude of an entity. In addition, if the entity is a town, the predicate informs the reader of the town's country of affiliation, if that information is contained in the database. *Location* also mentions all of the towns in the same block as the entity, and their distance and direction from the entity of interest.

Example

Message:
```
(LOCATION
    ((#<TOWN AS>)) ((#<CLASS TOWN>))
    ((LOCATION (CZ))
      (EXTERNAL-LOCATION (DEGREES-LONGITUDE 12
              NO-ARTICLE NIL DEGREES-LATITUDE 50)))
              ((NORTH-EAST KILOMETERS NO-ARTICLE
              ((#<CLASS TOWN> #<TOWN SELB>)) 7)))
```

Text: *As is a town located in Czechoslovakia at fifty degrees latitude twelve degrees longitude seven kilometers north-east of town Selb.*

Directional-Location

The *directional-location* predicate is a simple version of the *location* predicate. It just returns the latitude and longitude of an entity.

Example

Message:

```
(RELATIONAL-LOCATION
    ((#<TOWN MERSEBURG> #<CLASS TOWN>))
    ((DEGREES-LONGITUDE 12) (DEGREES-LATITUDE 51)))
```

Text: *The town Merseburg is located at twelve degrees longitude and fifty-one degrees latitude.*

Relational-location

Relational-location tells where one map-object is in relation to another map-object. It gives the latitude and longitude coordinates of the object of interest, and the distance and direction from the reference object.

Example

Message:

```
(RELATIONAL-LOCATION
    ((#<TOWN FREIBERG>)) ((#<CLASS TOWN>))
    ((LOCATION (GC))
        (EXTERNAL-LOCATION (DEGREES-LONGITUDE 13
        NO-ARTICLE NIL DEGREES-LATITUDE 51)))
        ((SOUTH-EAST KILOMETERS NO-ARTICLE
            ((#<CLASS LAKE> #<LAKE AATALSPERRE>)) 344)))
```

Text: *Freiberg is a town located in East Germany at fifty-one degrees latitude thirteen degrees longitude 344 kilometers south-east of lake Aatalsperre.*

Relational-point-location

This predicate uses the function *determine-closest-point* to find the point closest to the place of interest that is located on a road. This function *determine-closest-path* is explained in greater detail in Section 8. Once the closest point on a road is determined, the *relational-point-location* predicate gives the distance and direction of the place of interest in relation to this point.

Example

Message:

```
(RELATIONAL-POINT-LOCATION
((#<LAKE FRIEDERSEE> #<CLASS LAKE>))
NIL
   ((EXTERNAL-LOCATION (NORTH KILOMETERS NO-ARTICLE
   ((THIS POINT)) 2))))
```

Text: *The lake Friedersee is located two kilometers north of this point.* [*this point* refers to the closest point to the lake that is located on a road]

Illustration-by-logical-definition

This predicate illustrates an entity by listing its parents, and the attributes which make it unique from the other entities in the same class.

Example

Message:

```
(ILLUSTRATION-BY-LOGICAL-DEFINITION
   ((#<CLASS LAND-ROLL>))
   ((#< CLASS RADAR>))
   NIL
   NIL
   ((FREQUENCY 4 INDEF NIL NIL (G H I J))
    (VERTICAL-BEAMWIDTH NIL INDEF ((1 DEGREE)))
    (HORIZONTAL-BEAMWIDTH NIL INDEF ((3 DEGREES)))))
```

Text: *A land-roll, for example, is a radar with four band designators (g, h, i, and j), a vertical beamwidth of one degree, and a horizontal beamwidth of three degrees.*

Classification-illustration

Classification-illustration illustrates an entity by first listing its parents. Then if the entity has any offspring, they are listed as well.

Example 1

Message:

```
(CLASSIFICATION-ILLUSTRATION
   ((#<CLASS RADAR>))
   ((#<CLASS SENSOR>) (#<CLASS SIMULATION-ICON>)))
```

Text: *A radar, for example, is a sensor and a simulation-icon.*

Example 2

Message:

```
(CLASSIFICATION-ILLUSTRATION
   ((#<CLASS FIGHTER>))
   ((#<CLASS AIRCRAFT))
NIL
((#<CLASS YAK-28>)(#<CLASS SU-19>)
 (#<CLASS SU-17>) (#<CLASS SU-9>)
 (#<CLASS SU-7B>) (#<CLASS MIG-31>)
 (#<CLASS MIG-29>) (#<CLASS MIG-27>)
 (#<CLASS MIG-25>) (#<CLASS MIG-23>)
 (#<CLASS MIG-21>) (#<CLASS MIG-17>)
 (#<CLASS F-111>) (#<CLASS F-16>)
 (#<CLASS F-16>) (#<CLASS F-15>)
 (#<CLASS F-5E>) (#<CLASS F-4>)
 (#<CLASS ATTACK-AIRCRAFT>)))
```

Text: *A fighter, for example, is an aircraft such as a YAK-28, a SU-19, a SU-17, a SU-9, a SU-7b, a MIG-31, a MIG-29, a MIG-27, a MIG-25, a MIG-23, a MIG-21, a MIG-17, an F-111, an F-16, an F-15, an F-5e, an F-4, and an attack aircraft.*

Analogy

Analogy takes two entities and initially states that the first one is like the second one. It then calls *non-analogous-properties*, which compares the common attributes of the two entities and finds the common attributes with different values. It lists these values for the first entity.

Example

Message:

```
(ANALOGY
   ((#<TOWN GOTTINGEN> #<TOWN MOST>))
   NIL
   NIL
   ((LOCATION NIL INDEF ((10 DEGREES-LONGITUDE)
    (52 DEGREES-LATITUDE)))
    (POPULATION NIL INDEF 100000-OR-MORE)
    (ROADS  2  NIL  (#<road-segment...))))
```

Text: *Gottingen is like Most, however Gottingen has a location of ten degrees longitude and fifty-one degrees latitude, a population of 100000-or-more, and two roads (Route 3 and Route 27).*

Compare-Contrast

This predicate compares the values of the attributes of two entities using the function *comp-cont*. *Comp-cont* checks the equality of various attribute values, and tells whether or not the values are the same or different.

Example

Message:

```
(COMPARE-CONTRAST
    ((#<CLASS THIN-SKIN>) (#<CLASS FLAT-FACE>))
    ((CLASS SAME DEF) (FREQUENCY DIFFERENT INDEF)
     (MAP-DISPLAY SAME DEF)))
```

Text: *A thin-skin and a flat-face have the same class, a different band designator, and the same map-display.*

Inference

Inference takes two entities and establishes the relationship between them using the function *infer*. *Infer* checks the similarities of the classes of the two entities as well as the attributes common to both of them. It concludes that the two entities are either similar or different based on the similarities of their classes and attributes.

Example 1

Message:

```
(INFERENCE
    ((#<CLASS F-16>) (#<CLASS YAK-28>))
    ((CLASS SIMILAR NO-ARTICLE)))
```

Text: *Therefore, an F-16 and a YAK-28 are similar classes.*

Example 2

Message:

```
(INFERENCE
    ((#<TOWN BERLIN>) (#<CLASS WATERWAY>))
    ((OBJECT DIFFERENT NO-ARTICLE)))
```

Text: *Therefore, Berlin and a waterway are different objects.*

Comparison

Comparison is the exact same predicate as *inference*, however *inference* adds a "therefore" at the beginning of the sentence and *comparison* does not.

Example 1

Message:

```
(COMPARISON
    ((#<LAKE BECKEN>) (#<LAKE FRIEDERSEE>))
```

```
        ((INSTANCES SIMILAR NO-ARTICLE)))
```

Text: *Becken and Friedersee are similar instances.*

Example 2

Message:
```
(COMPARISON
    ((#<CLASS TOWN>) (#<CLASS RADAR>))
    ((CLASS DIFFERENT NO-ARTICLE)))
```

Text: *A town and a radar are different classes.*

Compare-similar-attributes

Compare-similar-attributes informs the reader of the availability of any common-valued attributes between two entities. It uses the function *same-values?* to return a list of common-valued attributes and their values. If the entities don't have any common-valued attributes or any same-valued attributes, *compare-similar-attributes* will concede that fact.

Example 1

Message:
```
(COMPARE-SIMILAR-ATTRIBUTES
    ((BOTH THEY NO-ARTICLE))
    ((STATUS NIL INDEF FULLY-FUNCTIONAL)
     (RUNWAY LARGE INDEF NIL)))
```

Text: *They both have a functional-status of fully-functional and a large runway.*

Example 2

Message:
```
(COMPARE-SIMILAR-ATTRIBUTES
    ((THEY)) ((ATTRIBUTES COMMON NO-ARTICLE NIL NO)))
```

Text: *They have no common attributes.*

Example 3

Message:
```
(COMPARE-SIMILAR-ATTRIBUTES
    ((THEY)) ((ATTRIBUTES IDENTICAL  NO-ARTICLE NIL NO)))
```

Text: *They have no identical attributes.*

Compare-different-attributes

This predicate returns the attributes that are common to two entities but have different values. It uses the function *common-properties* to find these attributes. It then calls *list-different-values*, which lists each set of different values and puts each list in a list along with the name of that particular attribute.

Example

Message:
```
(COMPARE-DIFFERENT-ATTRIBUTES
    ((#<CLASS RUNWAY> #<RUNWAY PEENEMUNDE-RUNWAY-14532S>)
     ((#<CLASS RUNWAY> #<RUNWAY PUTNITZ-RUNWAY-07S25S>))
    ((AFFILIATION DIFFERENT NO-ARTICLE NIL NIL NIL (((GC))((UR))))
     (AIR-FACILITY DIFFERENT NO-ARTICLE NIL NIL NIL
        (((AIR-FACILITY PEENEMUNDE)) ((AIR-FACILITY PUTNITZ))))
     (LENGTH DIFFERENT NO-ARTICLE NIL NIL NIL
        (((2.744 KILOMETERS)) ((3.201 KILOMETERS))))))
```

Text: *However, Peenemunde-Runway-14532s runway and Putnitz-Runway-07s25s runway have different affiliations (East Germany versus Soviet Union), different air facilities (Peenemunde versus Putnitz), and different lengths (2.744 kilometers versus 3.201kilometers).*

Entity-class-comparison

Entity-class-comparison relates two entities by stating that they are both members of a certain class. This class is found by using the function *compare-search-lists*, which retrieves the lowest-level common class in the two entities' search lists.

Example 1

Message:
```
(ENTITY-CLASS-COMPARISON
    ((CLASS #<CLASS SU-9>) (CLASS #<CLASS F-111>))
    ((#<CLASS FIGHTER> NIL NO-ARTICLE)))
```

Text: *An SU-9 and a F-111 are both fighters.*

Example 2

Message:
```
(ENTITY-CLASS-COMPARISON
    ((#<TOWN TAUCHA>) (#<WATERWAY EUROPA-CANAL>))
    ((CLASS #<CLASS CARTO-OBJECT> NIL NO-ARTICLE)))
```

Text: *Taucha and Europa-Canal are both carto-objects.*

However-comparison

This predicate compares two entities by combining their definitions into one sentence. The arrangement is similar to that of the *definition* predicate.

Example 1

Message:
```
(HOWEVER-COMPARISON
    ((#<TOWN APOLDA> NIL NO-ARTICLE)
     (#<AIRSTRIP STOD> NIL NO-ARTICLE))
    (((#<CLASS TOWN>)) ((#<CLASS AIRSTRIP>))))
```

Text: *However, Apolda is a town and Stod is an airstrip.*

Example 2

Message:
```
(HOWEVER-COMPARISON
    ((#<CLASS SAM> NIL NO-ARTICLE)
     (#<CLASS LAKE> NIL N0-ARTICLE))
    (((#<CLASS GROUND-VEHICLE>)) ((#<CLASS CARTO-OBJECT>))))
```

Text: *However, a surface-to-air missile is a ground-vehicle and a lake is a carto-object.*

Point-by-point-comparison

Point-by-point-comparison compares the properties that two entities have in common. It compares them one at a time, using the list of attributes sent from the function *common-properties*. The rhetorical predicate also uses a random generator to choose between "whereas" or a semicolon as a contrasting connective.

Example 1

Message:
```
(POINT-BY-POINT-COMPARISON
    ((#<HELIPORT DRESDEN-HELLERAU>) (#<AIRSTRIP GERA>))
    ((STATUS NIL INDEF FULLY-FUNCTIONAL)) NIL NIL
     ((STATUS NIL INDEF PARTIALLY-FUNCTIONAL)))
```

Text: *Dresden-Hellerau has a functional-status of fully-functional whereas Gera has a functional-status of partially-functional.*

Example 2

Message:

```
(POINT-BY-POINT-COMPARISON
    ((#<HELIPORT DRESDEN-HELLERAU>) (#<AIRSTRIP GERA>))
    ((LOCATION NIL INDEF ((14 DEGREES-LONGITUDE)
     (51 DEGREES-LATITUDE)))) NIL NIL
     ((LOCATION NIL INDEF ((12 DEGREES-LONGITUDE)
                                (51 DEGREES-LATITUDE)))))
```

Text: *Dresden-Hellerau has a location of fourteen degrees longitude and fifty-one degrees latitude; Gera has a location of twelve degrees longitude and fifty-one degrees latitude.*

The rhetorical predicate section presents a basis for the remaining sections of this report. It provides the format for each predicate's message, as well as an example of the type of text it can produce. The remainder of this report will explain how TEXPLAN is able to use the rhetorical predicates and their messages to generate multi-sentence English text. The next few sections relate how these rhetorical predicate messages can be manipulated by the system to actually generate the type of sentences illustrated in this section.

SECTION 2

THE TRANSLATE FUNCTION

The function *translate* takes the message organized by a rhetorical predicate and prepares it for realization to surface form. It accomplishes this through four levels of processing: pragmatics, semantics, relational-associations, and syntax. These four levels are realized in the functions *assign-pragmatic-function*, *assign-semantic-function*, *assign-relational-function* and *assign-syntax-function*, respectively. Each of these functions are detailed below.

Assign-Pragmatic-Function

Pragmatic analysis is performed by the procedure *assign-pragmatic-function*. This function considers focus and context to make pronominalization decisions when realizing multi-sentence text. However, since focus and context were not addressed by this work, this function does not return a value (see Maybury (1990) for a detailed discussion of focus and text generation).

Assign-Semantic-Function

Semantic analysis of the message is performed by *assign-semantic-function*. This function basically distinguishes the special types of lists that may occur in the fourth position of the message. These four special types of lists refer to phrasal adjuncts and include *instrument, location, function,* and *external-location.* If one of the lists in the fourth position of the message belongs to one of these categories, special steps must be taken in the realization of the surface form for that particular list. The keyword "instrument" signals that "with" should be inserted before the realization of that particular list. "Function" is the keyword that signals that "for" should be inserted before the realization. "Location" signals that the words "located in" should be inserted. Finally, "external-location" signals that either "located", "located at", or "at" should be used. The choice in this case depends on the value of the fourth position of that particular list. *Assign-semantic-function* also chooses the verb for the sentence. This is done by the function *rp-action.* This function looks up the "action" or verb that corresponds to the rhetorical predicate that it is trying to translate. For example, the rhetorical predicate *attributive* relies on the verb "have."

Assign-Relational-Function

After assigning semantic roles to the rhetorical predicate constituents, the function *assign-relational-function* generates the different structures of a sentence. It first calls the functions *make-pp, make-v* and *make-np*, which generate the prepositional phrases, verb phrases and noun phrases of a sentence, respectively (see Section 3). *Assign-relational-function* then calls the function *insertions.* This function inserts one or more connectives into a sentence, basing its choice of connective on the rhetorical predicate being used. The actual insertion would be performed by a specific function, called by *insertions.* For example, the usage of the rhetorical predicate *inference* would signal the function *therefore-insertion*, which would then insert a "therefore," at the beginning of the sentence.

Assign-Syntax-Function

As soon as the proper sentence structure is formed, *assign-syntax-function* takes a list of all of the word entries for a sentence and returns the surface form of each word along with its syntax. This function can be very confusing, due to the fact that unfortunately not all words have the same list formats. Most word entries are in one list, but there are some word entries that are embedded in three lists. *Assign-syntax-function* first checks to see if a word is in three lists. If so, it "takes" the word out of the lists, retrieves the surface form and syntax of the word, and recurses on the rest of the words in the original list. In addition, sometimes there are two words embedded together in 3 lists (like a value listed with its unit of measure). If this is the case, the function will individually include each word in the final function output. Also, a specific combination of *list* and *append* functions were needed to make the function run each time it recursed. The final function output consists of a list of lists; each individual list containing the surface form of a word and its syntactic features.

Having detailed how the *translate* functions are used to transpose the rhetorical predicate messages into English surface form, we now focus on how the system actually interprets the organization of the messages to realize grammatically correct and meaningful sentences.

SECTION 3

SENTENCE SYNTAX AND SEMANTICS

Sentence generation in TEXPLAN begins with a message or a list representing a sentence. This message or list is built using a rhetorical predicate (see Section 1). This structure represents a sentence and its components, which can include subjects, articles, direct objects, indirect objects and modifiers. This structure is versatile: by using different forms of the components or by suppressing a component in the message, different types of sentences can be generated. The message consists of seven positions which may contain either a value or "nil". These positions, each of which will be detailed in turn, represent the subject, direct object, article, prepositional phrases, quantifiers, and descriptions of direct objects.

The first position is the grammatical position for the subject. The value of this slot may be either a list, or a list of lists. There are four basic forms that this value may take:

1. Simple subject	((subj))
2. Modified subject	((subj mod))
3. Compound subject	((subj1) (subj2))
4. Modified compound subject	((subj1 mod1) (subj2 mod2))

The second position in the list is reserved for the direct object. Its value may also be either a list or a list of lists. It too has four forms that are similar to the forms of the subject slot.

The third position in the list represents the article to be used with the subject. The value of this slot must equal one of four keywords:

1. indef	-	a, an
def	-	the
3. no-article	-	suppress article placement
4. nil	-	use the default value of this slot

The fourth position is used for prepositional phrases that refer back to the subject. The preposition associated with this slot is the word "of". The value of the slot must be a list of lists; a list being comprised of a prepositional object and an optional modifier. In the case of more than one list, multiple prepositional objects will be created.

The fifth position is the slot for prepositional phrases that refer back to their own object. The preposition associated with this slot is "with". The layout of this slot is similar to the overall layout of the message, such as:

(((subject)) ((direct-object)) article (simple-prepositions))

This slot is primarily used when generating a sentence that tells the user about an entity's attributes and their values. For example, the message for defining town Freiberg (using the rhetorical predicate *logical-definition*) would be:

```
(LOGICAL-DEFINITION
   ((#<TOWN FREIBERG>))
   ((#<CLASS TOWN>))
   NIL
   NIL
   ((LOCATION NIL INDEF ((13 DEGREES-LONGITUDE)
      (51 DEGREES-LATITUDE))) (STATUS NIL INDEF FULLY-FUNCTIONAL)
   (POPULATION NIL INDEF 25000-TO-100000) ROADS 2 NIL
      (#<road-segment 173 33UUS8441 33UUS8041>
      #<road-segment 101 33UUS8440 33UUS8441>
      #<road-segment 101 33UUS8441 33UUS8046>
      #<road-segment 173 33UUS9049 33UUS8441>))))
```

Here, the relevant attributes of Freiberg (location, status, population, and roads) are listed in the fifth position of the message along with their values. The value in this slot must be a list of lists. Multiple lists will create multiple prepositional phrases which refer back to their own object.

The sixth position is the grammatical slot for a quantifier. In the surface form, the quantifier is placed before the subject and before any modifiers of the subject. The quantifier may include words like "few", "some", "all", and "most".

The seventh position in the message is the slot for a list which follows the direct object. The purpose of this list is to clarify and describe the direct object, usually by listing its parts. The value of this slot must be a list. The surface form is similar to the slot value, except for the insertion of commas and the word "and" when there are multiple elements in the list.

The manipulations of this message are performed by the functions *make-np*, *make-pp* and *make-v*, each of which are described below. They are all called by the function *assign-relational-structure* (see Section 2).

Make-np

The function *make-np* forms noun phrases. It inserts conjunctions, selects determiners, pluralizes words, and selects the necessary punctuations for the final surface form. *Make-np* then arranges the words in the correct grammatical order for final realization.

Make-pp

The function *make-pp* produces prepositional phrases. It is called by *assign-relational-structure* for each indirect object to be used in the sentence. Each indirect object has a specific preposition which is used to generate the prepositional phrase. This preposition is sent to *make-pp* by *assign-relational-structure* and is returned by *make-pp* as part of a prepositional phrase.

Make-v

The function *make-v* returns a verb phrase. It selects the proper form of a verb based on its tense ("past", "present" or "future") and its voice ("passive" or "active"). For example, given (*be present active*) as the arguments, *make-v* would return the different present active forms of the verb "be" (i.e. "is", "are", "am").

Having detailed the message structure and how it is manipulated to produce a final grammatical surface form, we now turn to a description of the lexical entries that make up these messages; the entries that represent every word and symbol used by the system.

SECTION 4

THE DICTIONARY

The dictionary refers to all of the lexical entries in TEXPLAN, which includes the entries explicitly listed in the dictionary file as well as the entries automatically generated by the system. The dictionary file consists of entries of the most frequently used words and symbols in TEXPLAN, including verbs, adverbials, prepositions, and words that possess a specific surface form that cannot be automatically generated. For example, the final surface form of the acronym "SAM" is not "SAM", but "surface-to-air missile." The generated entries for words not listed explicitly in the dictionary file are formed by specific functions. Entries in the dictionary file are grouped according to their parts of speech. Nouns are in one section, verbs in another, and so on. For each new dictionary entry that is manually added by the user, the function *make-dictionary-entry* must be run with the new entry as the argument. This will place the new entry on the top of the stack of old entries. If the word is already in the dictionary, the most recent entry will be used by the system. The following subsections describe the different functions used to generate lexical entries.

Look-up

Look-up is a function which returns a specific dictionary entry for each word and symbol used in the final text. The dictionary entry of a word guides the sentences generator in incorporating the word into a sentence. A dictionary entry is a list consisting of the word and its syntax, semantics and realization. The syntax of a word varies, depending on its part of speech. In general, however, it includes the part of speech, agreement and morphological information. Semantics include a logical form meaning representation of the lexical item. Realization is the final generated English form of the word. For example, the dictionary entry for the word "town" is:

(town ((noun count sing neuter) town "town")

where *town* is the word given, *noun* is the part of speech, *count* is the type of noun it is (either mass noun or count noun), *sing* refers to the word's singularity (*plur* would mean the word is plural), *neuter* is the gender of the word, *town* is the semantic form, and "town" is how the word would be printed in the final text.

The *look-up* function first considers whether or not a word has a lexical entry already stored into the dictionary. If it does, it returns it. If it does not, it automatically generates one. *Look-up* will automatically generate lexical-entries for numbers, since numbers do not have entries previously stored into the dictionary. *Look-up returns* the extended written form of a number for numbers less than 100 (i.e. ninety-nine), and using the numerical representation of a number for numbers greater than or equal to 100 (i.e. 101).

If the word is a LACE or MAP object, *look-up* will call the function *dictionary-entry-of-LACE-object* (see below). This function returns a dictionary entry of a word if one exists, otherwise it generates one. If the word does not fit into a specific category (like a number or a LACE or MAP object), *look-up* assumes that it is a noun, and generates a noun entry.

Dictionary-entry-of-LACE-object

Dictionary-entry-of-LACE-object is a function which actually builds a dictionary entry for a LACE or MAP object that does not already have an entry. If the object is a class object, and it does not already have an entry in the dictionary, the function returns an entry with noun syntax, using the object's print-name as its final surface form. For example, the dictionary entry for *#<CLASS RADAR>* is:

```
(#<CLASS RADAR> ((noun count sing neuter) #<CLASS RADAR> USER::RADAR))
```

If the object is an OCA mission (Offensive Counter Air Mission), then the function returns the proper-noun syntax, because this a uniquely-named individual. If the object is an instance, then the function *instance-entry* is called (see below). *Instance-entry* generates lexical entries for specific instances. If the object does not fall under any of these three categories, the function assumes that it is a noun, and returns a noun entry.

Instance-entry

Instance-entry distinguishes between the different types of instances in the LACE and MAP systems. Different types of instances require different methods to build their proper dictionary entries. For example, roads and intersections are both MAP objects but they have distinct surface forms (i.e. "Route 173" versus "the intersection of Route 97 and Autobahn E63"). If the instance is not one of the types of instances tested for in the function, it is assumed to be a proper noun, and an entry for a proper noun is returned.

This explanation of how the lexical entries are formed completes the discussion of the linguistic realization component of TEXPLAN. What follows is an explanation of the text planning constituent of the system. These subsequent sections address the issue of how TEXPLAN selects and orders sequences of rhetorical predicates to generate a coherent multi-sentence text that realizes a specific goal.

SECTION 5

THE TEXT PLANNER

The purpose of the text planner is to combine various rhetorical predicates into a paragraph that achieves a particular goal. The planner reasons about communicative actions (represented as plan operators) to produce a hierarchial text plan that can be executed to accomplish a goal. Each plan operator is make up of seven slots: name, header, constraints, preconditions-essential, preconditions-desirable, effects, and decomposition (described below). Multiple plan operators may exist that accomplish the same goal; however, each operator achieves it by different means. When asked to plan text, the planner selects the first plan operator that will accomplish the goal. If this plan meets all of the essential preconditions and the parameters are within the domain of that particular plan, then the planner uses that plan to generate the text.

The **name** slot is a short English description of what that particular plan operator does. Every plan operator has a unique name slot.

The **header** slot identifies the name of the rhetorical act and the parameters involved in the action defined by the plan operator. Multiple plan operators may have the same headers, since there may be multiple ways to accomplish one high-level action (i.e. rhetorical act).

The purpose of the **constraints** slot is to guide the selection of plan operators. Some plans will not work if given a particular class or if the object does not have specific characteristics. However, in the LACE application, many of the plan operators work for all object parameters, so the **constraints** slots are set to the default value of “t”.

The **preconditions-essential** slot helps select among different plan operators that accomplish the same goal and that satisfy the **constraints** slot. This slot establishes conditions on the parameters, and if these conditions are not met, the next plan operator that accomplishes the same goal will be selected. The value of this slot ensures that the plan operator that best accomplishes the goal will be selected.

The **preconditions-desirable** slot is currently set to “t” for all the plan operators. It is intended to be used in the future for further decomposition of conditions for better planning of text.

The **effects** slot indicates what the action characterized by the plan operator will achieve if executed. It describes the changes in the intended knowledge, beliefs and desires of the hearer. In this way, as text is generated, the system can record a list of expected changes in the cognitive state of the hearer.

The **decomposition** slot actually breaks down the communicative action captured by the plan operator into lower-level actions. These actions consists of illocutionary speech acts (e.g. inform, request), which call rhetorical predicates, and rhetorical acts which include acts such as describe, define, compare, and enable. These rhetorical acts invoke other plan operators.

Now that the overall organization of the text planning component of the system has been explained, the final few sections will describe the different plan operators that have been built and/or reconstructed for use in the LACE application.

SECTION 6

THE DEFINE AND DESCRIBE PLANNERS

TEXPLAN uses two sets of plan operators, *describe* and *define* plan operators, to give descriptions of entities in the system. The *describe* and *define* planners are interdependent; the second is a component of the first. They have very similar structures and often result in similar descriptions. The *describe* and *define* operators also share the same versatilities in their description options.

There are eight working *describe* plan operators. However, only one of them will be detailed in this report since it was the only one used for the LACE application. This plan operator, *describe-by-defining-entity*, is shown in Figure 6.1. The **header** slot shows the name of the rhetorical act (*describe*), along with the speaker and hearer notations, and the place reserved for the entity, illustrated in italics. This plan operator calls for the *define* rhetorical act, as illustrated in the plan operator's **decomposition** slot.

```
NAME:             'describe-by-defining-entity
HEADER:           '(describe S H _entity)
CONSTRAINTS:      '(and (entity? _entity) (or (HASTE S) (HASTE H)))
PRECONDITIONS-ESSENTIAL:
                  '(and (KNOW-ABOUT S _entity)
                    (WANT S (KNOW-ABOUT H _entity)))
PRECONDITIONS-DESIRABLE:  t
EFFECTS:          '(KNOW-ABOUT H _entity)
DECOMPOSITION:    '((define S H _entity)))
```

Figure 6.1. Describe-By-Defining-Entity Plan Operator

After *define* is called, the system sequentially tests the preconditions of each of the three different *define* plan operators until it finds a suitable definition method for its entity. For the purpose of the LACE application, this plan operator, *define-using-logical-definition* (shown in Figure 6.2), is the only one in use. Note that this is due to the fact that it has no preconditions, and will subsequentially always be chosen by the system.

```
NAME:                       'define-using-logical-dcfinition
HEADER:                     '(define S H _entity)
CONSTRAINTS:                t
PRECONDITIONS-ESSENTIAL:    t
PRECONDITIONS-DESIRABLE:    t
EFFECTS:                    nil
DECOMPOSITION:              '((inform S H (logical-dcfinition _entity))
```

Figure 6.2. Define-Using-Logical-Definition Plan Operator

The *logical-definition* rhetorical predicate used in this plan operator informs the hearer about an entity by describing its superclass and its distinguishing features. The two other *define* plan operators found in the text-plan file make use of the rhetorical predicate *synonymic-definition* (see Section 1), and a rhetorical predicate not yet written, *antonymic-definition*. Other *describe* plan operators in the file describe attributes, constituents, subclasses or instances of an entity, or motivate an illustration of a given entity.

Having provided an outline of the different *define* and *describe* plan operators, the next section deals with another group of operators; the *compare* plan operators.

SECTION 7

THE COMPARE PLANNER

While the *define* and *describe* plan operators are used to characterize one entity, the *compare* planner is used to make comparisons between two entities. There are three major types of comparison plan operators that allow this planner to diversify its method of comparison: *similarities/aifferences-comparison*, *point-by-point-comparison* and *compare-describe-in-turn*. The choice among these alternatives is guided by the essential preconditions in the plan operators. For example, the *similarities/differences-comparison* plan operator can only be used on two entities that have more than one common attribute with the same value and more than one common attribute with a different value. The essential preconditions of the *point-by-point-comparison* operator require that the two entities be different entities that have common attributes with different values. The third comparison operator, *compare-describe-in-turn*, has no essential preconditions. It is used as a last resort if the preconditions of the other two comparison plan operators fail.

The first type oı comparison plan operator, *similarities/differences-comparison*, can be broken down into three rhetorical predicates (rp). The system first informs the hearer of the inferred relationship between the two entities using the rp *comparison*. Next, it informs the hearer of the common-valued attributes of the two entities with the rp *compare-similar-attributes*. Finally, *compare-different-attributes* retrieves the different-valued characteristics of the entities. Each characteristic is then coupled with a list containing two values (the values of the characteristic for each entity). This decomposition can be seen in the actual *similarities/differences-comparison* plan operator shown in Figure 7.1.

```
NAME:           'similarities/differences-comparison
HEADER:         '(compare S H _entity1  _entity2)
CONSTRAINTS: t
PRECONDITIONS-
 ESSENTIAL:     '(and  (> (length  (common-properties-and-values
                                             (list _entity1   _entity2))) 1)
                       (> (length  (common-properties-1  (list  _entity1   _entity2))) 1))
 DESIRABLE:  t
EFFECTS:        nil
DECOMPOSITION:  '((inform  S  H  (comparison  _entity1   _entity2))
                  (inform  S  H  (compare-similar-attributes  _entity1    _entity2))
                  (inform  S  H  (compare-different-attributes  _entity1    _entity2))))
```

Figure 7.1. Similarities/Differences-Comparison Plan Operator

For example, TEXPLAN uses this plan operator to compare a MIG-27 and a MIG-29, which generates the following output:

> *A MIG-27 and a MIG-29 are similar fighters. They both have a maximum fuel capacity of 2,000, 233.0 meters per pound of fuel, and a maximum speed of five meters per second. However, the MIG-27 fighter and the MIG-29 fighter have different nick-names (Flogger-D versus Fulcrum), different cruise-speeds (490 meters-per-second versus 660 meters-per-second), different offensive capabilities (very-weak versus very-strong), different defensive capabilities (strong versus weak), and different combat radii (1,380 kilometers versus 1,200 kilometers).*

The next type of comparison plan operator is *point-by-point-comparison*. The *point-by-point-comparison* plan operator uses four rhetorical predicates. *Entity-class-comparison* informs the hearer of the lowest-level class that the two entities have in common. The system next uses the rp *however-comparison* to inform the hearer of which class each object is from. The third rhetorical predicate, *point-by-point-comparison*, compares each of the objects' different-valued attributes. This is done using a *mapcar* type of macro (*for-all*) to run the rp on each attribute. The fourth rp is *compare-similar-attributes*. Its usage in this plan operator can be misleading, since the two objects being compared by the point-by-point-comparison have no similar attributes. However, *compare-similar-attributes* simply informs the hearer of this fact; it doesn't actually try to compare any attributes. The *point-by-point-comparison* plan operator is illustrated in Figure 7.2.

```
NAME:          'point-by-point-comparison
HEADER:        '(compare S H _entity1 _entity2)
CONSTRAINTS: t
PRECONDITIONS-
 ESSENTIAL:  '(and  (eq (infer (list _entity1  _entity2)) 'different)
                    (and (> (length  (common-properties (list _entity1  _entity2))) 1)
                         (eq  (common-properties-and-values
                                          (list _entity1   _entity2)) nil)))
PRECONDITIONS-DESIRABLE:  t
EFFECTS:       nil
DECOMPOSITION:
               '((inform  S  H  (entity-class-comparison _entity1  _entity2))
                 (inform  S  H  (however-comparison _entity1  _entity2))
                 (for-all  (common-properties  (list   _entity1   _entity2))
                    (inform  S  H  (point-by-point-comparison (_entity1  _entity2  _var))))
                 (inform  S  H  (compare-similar-attributes  _entity1  _entity2))))
```

Figure 7.2. Point-by-Point-Comparison Plan Operator

An example of this type of comparison using the town Freiberg and the lake Becken (from the LACE application) is:

> *Both Freiberg and Becken are carto-objects. However, Freiberg is a town and Becken is a lake. Freiberg has a location of thirteen degrees longitude and fifty-one degrees latitude whereas Becken has a location of eleven degrees longitude and fifty-one degrees latitude. Freiberg has a functional-status of partially-functional; Becken has a functional-status of fully- functional. They have no identical attributes.*

The final type of comparison plan operator is *compare-describe-in-turn*, which is based upon four rhetorical predicates. The system first informs the hearer of the logical definition of the entity using the rp *logical-definition*, which defines the first entity and describes all of its relevant attributes. The second rp, *compare-logical-definiton*, performs the same function as *logical-definition*, but *compare-logical-definition* also infers the relationship between two entities and relays this to the hearer, using a cue word like "similarly" or "in contrast", while at the same time conveying the fact that it is comparing the second entity to the first. It then uses the rp *compare-contrast* to compare all of the similar attributes of the two entities (both equal and different-valued). Finally, *inference* concludes that the entities are either similar or different, based on a comparison of their attributes. This comparison technique is formalized in the plan operator shown in Figure 7.3.

```
NAME:          'compare-describe-in-turn
HEADER:        '(compare S H _entity1 _entity2)
CONSTRAINTS:'(and (entity? _entity1) (entity? _entity2))
PRECONDITIONS-ESSENTIAL:  'none
PRECONDITIONS-DESIRABLE:  t
EFFECTS:       nil
DECOMPOSITION:
               '((inform S H (logical-definition _entity1))
                 (inform S H (compare-logical-definition _entity2))
                 (inform S H (compare-contrast _entity1 _entity2))
                 (inform S H (inference _entity1 _entity2))))
```

Figure 7.3. Compare-Describe-In-Turn Plan Operator

Comparing two towns, Merseburg and Erfurt, which are represented in the LACE application, this plan operator would generate the following paragraph:

> *Merseburg is a town with a location of twelve degrees longitude and fifty-one degrees latitude, a functional-status of fully-functional, a population of 25000-to-10000, and two roads (Route 181 and Route 91). Similarly, Erfurt is a town with a location of eleven degrees longitude and fifty-one degrees latitude, a functional-status of fully-functional, a population of 10000-or-more, and five roads (Route 7f, Route 5f, Route 274f, Route 7 and Route 4). Merseburg and Erfurt have the same class, a different location, the same functional-status, a different population, and several different roads. Therefore, they are similar instances.*

Having shown how TEXPLAN is capable of diversifying its comparisons of two entities found in the LACE application, we now detail the methods used to provide travel directions from a town in the application to another object in the LACE system.

SECTION 8

THE ENABLE-GO PLANNER

Just as it is able to inform the hearer about entities using descriptions and comparisons, TEXPLAN can also inform the hearer how to travel between two points. These locational instructions (i.e. route plans) are produced by the *enable-go* plan operator, *enable-to-get-to*, which is capable of giving two types of directions: directions to objects connected by roads (such as towns or intersections) and directions to objects not connected by roads (such as lakes, dams or any object in the Map Display System that has a location). Both types of directions are introduced by the *relational-location predicate* which tells the user where the place of destination is in relation to the point of origin. This particular predicate provides the latitude and longitude coordinates of the destination point, the distance to the destination point, and the direction of the destination point in relation to the origin. Next, a point-by-point route is given. The route-planner that is currently being used, called *find-decent-path*, is the same route-planner used by LACE to move mobile SAM sites. It was designed to find a relatively short path in a reasonable amount of time. Once the path is found, it is given to *genny-path*, a function which condenses the path to eliminate segments that are components of the same road, and adds the final destination as the last element of the list. This is illustrated in Figure 8.1 for a short path:

Input: (intersection<#> road-segment 57<#> road-segment 57<#> intersection<#>)

↓

Function: *genny-path*

↓

Output: (intersection<#> road-segment 57<#> intersection<#> town<#>)

Figure 8.1. Genny-Path Function Input/Output

The path is given to *go-event-predicate* which in turn calls *go-event-town*, *go-event-intersection*, *go-event-road*, or *go-event-true-destination* (used only when the destination object is not connected by roads) depending on the value of the current-segment. *Go-event-predicate* updates the local spatial focus of attention (i.e. the current entity in the space being discussed). There are several variables that trade the spatial focus, including *previous-segment*, *current-segment*, *next-segment*, *next-next-segment*, *final-segment*, and *destination* (the last two will be the same only in the case of non-road-connected directions). This local spatial focus is necessary to determine the distance and direction traveled on a particular road and to pronominalize (e.g. "from here").

Another feature of the directions produced by TEXPLAN is that sentence introductions are randomly selected. Not only does this produce variety within the paragraph, but it also limits the chance that two identical requests for directions will result in the same response. The range of introductions are: "From here", "From that town", "From that intersection", and "From (*town-name*)". The directions also substitute "continue on" for "take" if they have gone through a town, but remain on the same road. These lexical variations make the directions less repetitive and more like the text a human would produce. The examples below illustrate this lexical variation:

Merseburg is a town located at fifty-one degrees latitude twelve degrees longitude one hundred thirty-two kilometers north-west of town As.

From As take Route 21 south-east five kilometers to the Intersection of Route 92 and Route 21.

At that intersection take Route 92 north-west thirty-six kilometers to Oelsnitz.

From Oelsnitz continue on Route 92 north-west thirty-eight kilometers to the Intersection of Route 175 and Route 92.

From that intersection take Route 2 north-east twenty-eight kilometers to the Intersection of Route 2 and Route 176.

From there take Route 176 north-west thirty-nine kilometers to the Intersection of Route 91 and Route 176.

At that Intersection take Route 91 north-east sixteen kilometers to Merseburg.

Directions to objects that are not connected by roads are basically the same as directions to objects that are connected by roads. However, since the route planner cannot determine a path to an object that is not connected by a road, TEXPLAN must give it a substitute for the destination. The best choice for this substitute is the closest intersection to the destination. *Give-closest-intersection* searches a thirty kilometer radius to find the closest intersection to the destination. This function is similar to the LACE function that sweeps the area around a SAM site for targets. It adds a small incremental distance at varying angles to get a locus of points around the destination point. It increments its sweeping distance until it reaches a maximum of thirty kilometers. The LACE function *find-block* is used to determine the map blocks that the points are located in. Then the ERIC command (*ask map-block recall your intersections*) returns all the intersections in a particular block. A distance function is used on each of the intersections to determine the closest intersection to the destination. The route planner is then asked to plan a route between the town of origin and this intersection. Next, TEXPLAN calculates the point found on one of the roads in the intersection that is the closest point to the destination, bringing the user closer to the destination. This is done by the *select-closest-point* function. First all the road-segments that have an endpoint at the intersection are found by using the ERIC command (*ask intersection recall your roads*). Second, each of these road segments are placed in a group which includes all of the road segments from the intersection closest to the destination to the next intersection on that particular road. Third, a geometrical algorithm is used on each of the road-segments. This geometrical algorithm finds a point on the road-segment that is the closest to the destination creating a perpendicular line from each road-segment to the destination, then finding the perpendicular line with the shortest distance (see Figure 8.2). It finds a perpendicular line to the destination, the total distance traveled from the intersection to the beginning of the perpendicular line, and the distance traveled along that particular road-segment to reach the destination point. Fourth, a sorting function is used to place the road-segment with the closest point to the destination in the last position of the list and determine the total distance traveled from the intersection to that point (adding up the length of each individual segment). These road-segments are placed in a new list and the sorting function is performed again across road-segment groups to get the final solution. The final solution is in the form:

(road-seg-name dist-trav-along-road-seg dist-from-destination dist-from-inter)

The predicate *go-event-true-destination* uses this information to tell the user the distance and direction the user must travel to the closest point to the destination.

The final predicate in this text-plan is the *relational-point-location* predicate. It uses the road-segment information to tell where the new point is in relation to the destination. Due to the fact that the point is only represented by a distance and a direction traveled along a road-segment, it must be converted into map coordinates to determine the distance and direction with respect to the destination. This is done by determining the change in the x-direction and y-direction of the road-segment, then determining the ratio of the length of the road-segment with the length of the portion of the segment traveled. By multiplying the [illegible] in the x-direction and y-direction, the change in these directions for the portion of the segment traveled can be determined. These changes are added to the road-segment end-point that is closest to the intersection to get the map-coordinates of the closest-point to the destination. An example of directions given to an object that is not connected by roads is:

Friedersee is a lake located at fifty-two degrees latitude twelve degrees longitude forty kilometers north-east of town Leipzig.

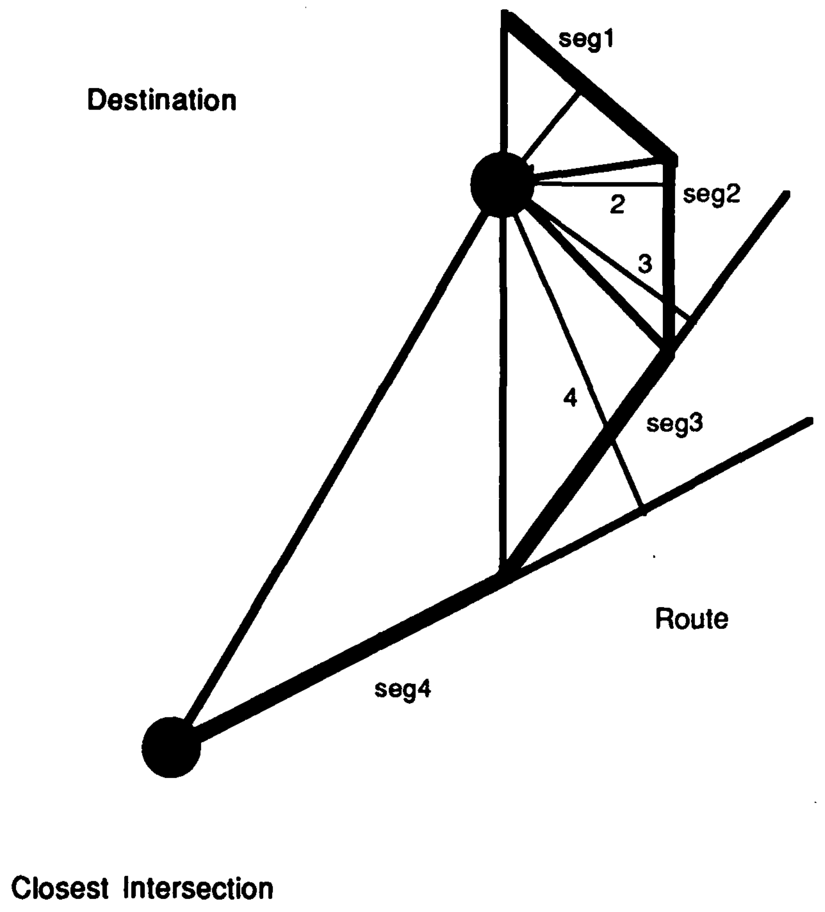
From Leipzig take Route 183 north-east thirty-four kilometers to the Intersection of Route 183a and Route 2.

From there continue on Route 2 north-east four kilometers to the Intersection of Route 107, Route 183 and Route 2.

At that intersection take Route 183 north-west eleven kilometers to the Intersection of Route 183 and Route 100.

From there take Route 183 west 4.0 kilometers.

The lake Friedersee is located two kilometers north of this point.



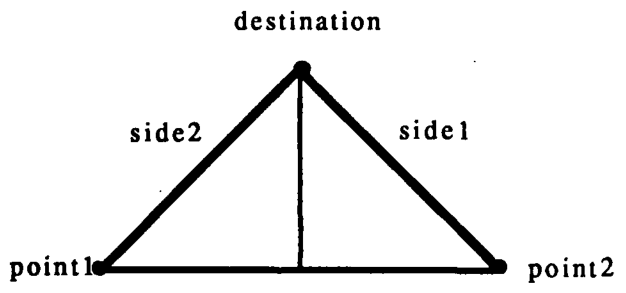


Figure 8.2: This figure shows how the closest point to the destination on a road segment is selected. A perpendicular line is drawn to each of the four road segments. Sometimes a line must be drawn to extend the road segment to find a perpendicular line. The function "select-closest-point" discards these cases by determining if the distance from the perpendicular line from the beginning point of the perpendicular line is greater than the length of the line segment.

CONCLUSION

This report is a brief overview of some of the components and capabilities of the TEXPLAN text generation system. The report breaks down the major components of the system, including the most important functions used in each component. However, the functions described in this report utilize more specific update and accessor functions to communicate with the underlying application. Although an explanation of these functions is beyond the scope of this report, a list of the most important and most used functions in TEXPLAN, along with their function calls, can be found in the appendix of this report.

A future goal of the work on TEXPLAN is to transport the system to LACE; to generate reports of the events that occur during the simulation. TEXPLAN has the capability to either give a description of the events in the simulation as they pertain to one object or as they pertain to the whole simulation. It is important to remember, however, that the simulation must be run before TEXPLAN can generate text.

This report has described three classes of text produced by TEXPLAN: descriptions, comparisons, and route directions. The system composes text by reasoning about the different kinds of information contained in the underlying application (e.g. LACE). Information is identified using rhetorical predicates which are realized as English text using an on-line dictionary and grammar. As illustrated in this report, this process ultimately results in coherent and cohesive English prose.

APPENDIX

COMMONLY USED FUNCTIONS IN THE TEXPLAN SYSTEM:

These functions are grouped according to their locations in the TEXPLAN (i.e. GENNY) directory. Each grouping represents a subdirectory and the file in which the function(s) can be found (underlined and in bold). Each function is listed with its argument(s) in italics. The functions used to run the system have their function calls listed as well.

Dictionary>lookup:

(look-up *object*)

(dictionary-entry-of-LACE-object *object*)

(instance-entry *object*)

Dictionary>makedictionary:

(make-dictionary-entry *entry*)

Planning>kb-interface:

(access-value *thing slot*)

(append-value *thing slot new-value*)

(update-value *thing slot value*)

Planning>lace-directions:

(go-event-predicate *path*)

(go-event-town *previous-segment current-segment next-segment next-next-segment final-segment destination*)

(go-event-intersection *previous-segment current-segment next-segment next-next-segment final-segment destination*)

(go-event-road *previous-segment current-segment next-segment next-next-segment final-segment destination*)

(go-event-true-destination *previous-segment current-segment next-segment next-next-segment final-segment destination*)

(give-closest-intersection *e*)

(select-closest-point *intersection road-segs destination*)

(closest-point-distances *intersection destination road-seg*)

Planning>planner-macros:

(genny-path *route*)

Planning>predicates:

(trans predicate-type *object*)

Trans is the abbreviated version of the function translate.

(instantiate-predicate *type object speech-act*)

Function call: (instantiate-predicate '*rhetorical-predicate object* 'define)

(attribute-value-pairs *object attribute-slots*)

(relevant-attributes *object*)

Realization>map-translate:

(translate *rp+focus+context*)

(assign-semantic-function *rp*)

(assign-relational-function *rp case pragmatics*)

(rp-action *rp*)

(insertions *relational-structure form voice*)

Realization>morphsyn:

(morph-syn *root entry*)

Realization>nogen:

(assign-syntax-function *lexical-list*)

Realization>relationalgram:

(make-np *agents focus context connective &optional (no-pronom nil*))

(make-nps *agents focus context connective &optional*

(suppress-pronominalization nil))

(make-all-nps *agents focus context connective &optional (no-pronom nil*))

(select-determiner *rp-skeleton entry context*)

Select-determiner will only take lists as arguments.

Realization>surface-form:

(surface-form *lex-punct-list*)

Function call: (surface-form (trans '*rhetorical-predicate object(s)*))

Maybury>genny>planning>planner:

(plan-and-display-text *effect*)

Function call: (trans '(*plan-operator* s h *object(s)*)))

The object in this function must be in the object form: #<TOWN FREIBERG>. If there are two objects, they must be listed: (list #<TOWN FREIBERG> <TOWN GOTTINGEN>).

HAL:>lace>ground-route>search:

(find-decent-path *start* *finish*)

REFERENCES

Anken, C. S. October, 1989. "LACE: Land Air Combat in Eric." Rome Air Development Center TR-89-219, Griffiss AFB, NY.

Hilton, M. L. and C. S. Anken. February, 1990 "Map Display System: An Object-Oriented Design and Implementation." Rome Air Development Center TR-90-54, Griffiss AFB, NY.

Hilton, M. July, 1987. "ERIC: An Object-Oriented Simulation Language." Rome Air Development Center TR-87-103, Griffiss AFB, NY.

Knowledge Systems Concepts, Inc. August, 1989. "Cooperative Red and Blue Database System." Final Technical Report, contract number F30602-87-D-0095. Prepared for Rome Air Development Center/COES, Griffiss AFB, NY.

Maybury, M. T. September 1990c. "Planning Multisentential English Text Using Communicative Acts." PhD Dissertation, Cambridge University Computer Laboratory: Cambridge, England.

BIBLIOGRAPHY

Maybury, M. T. August, 1989. “Knowledge Based Text Generation.” Rome Air Development Center TR-89-93, Griffiss AFB, NY.

Maybury, M. T. June, 1990a. “Using Discourse Focus, Temporal Focus, and Spatial Focus to Generate Multisentential Text.” Proceedings of the 5th International Workshop on Natural Language Generation, Linden Hall, Dawson, PA, 3-6 June, 1990.

Maybury, M. T. July, 1990b. “The Four Forms of Explanation Presentation: Description, Narration, Exposition, and Argument.” Proceedings of the AAAI-90 Workshop on Explanation, Boston Hynes Center, Boston, MA, July 30, 1990.

Maybury, M. T. September 1990c. “Planning Multisentential English Text Using Communicative Acts.” PhD Dissertation, Cambridge University Computer Laboratory: Cambridge, England.